Programmieren
mit
LISP

aus: *ComputerKurs*-Magazin, Jahrgänge 1984/1985

Listige Listen

Mit diesem Artikel beginnt eine Serie über LISP, in der wir die Anwendung dieser Sprache beschreiben und untersuchen, warum sie im Bereich der Künstlichen Intelligenz so weit verbreitet ist.

LISP wurde in den letzten Jahren hauptsächlich durch ihren Einsatz im Bereich der KI bekannt. Im Laufe der Zeit wurde dann deutlich, daß LISP sich als Allzwecksprache für einen breiten Anwendungsbereich eignet.

LISP taucht inzwischen in vielen sehr unterschiedlichen Bereichen auf. Unter LISP entstanden Betriebssysteme, Compiler und sogar Abenteuerspiele. Doch trotz der vielen LISP-Dialekte, die durch die weite Verbreitung der Sprache entstanden, halten sich die meisten LISP-Programme (aufgrund der einfachen und direkten Sprachstruktur) an einen Standard und lassen sich leicht von einer Maschine auf eine andere übertragen.

Beim Gebrauch von LISP wird man früher oder später bestimmte Funktionen und Befehle vermissen. Da ein „offizieller" Standard fehlt, haben die Programmierer der einzelnen Sprachversionen nur die Funktionen eingebaut, die ihnen wichtig erschienen. Die Sprache kann jedoch leicht um zusätzliche Befehle erweitert werden.

LISP unterscheidet sich grundlegend von bekannteren Sprachen wie PASCAL, FORTRAN und BASIC. Durch ihren sehr einfachen und einheitlichen Aufbau eignet sie sich ideal für Datenverarbeitung.

Die Listenstruktur der Sprache läßt sich leicht an die wesentlichen Informationsstrukturen des jeweiligen Computers anpassen. LISP selbst kann dabei sortieren, suchen, arithmetische Funktionen und sogar Spiele ausführen. Mit LISP (insbesondere mit dem in dieser Serie vorgestellten Acornsoft-LISP für den Acorn B) lassen sich außerdem die speziellen Sound- und Grafikmöglichkeiten der meisten Microcomputer ansprechen. Unsere Beispiele lassen sich auf andere Sprachversionen übertragen.

Die Grundlage von LISP ist die Datenstruktur der „Liste". (Der Name LISP entstammt dem Begriff „list processing" – Listenverarbeitung.) Listen ähneln in mancher Hinsicht den vertrauten Arrays. Im Gegensatz zu Arrays haben Listen jedoch keine festgelegte Länge.

Jede Liste von Elementen kann von Klammern umschlossen werden:

```
(a b c d e . . .)
```

wobei a, b, c etc. die Bestandteile der Listen sind. Die einzelnen Teile werden dabei „Atome" genannt. Sie können Zahlen, nichtnumerische Daten (Zeichen oder Variablen) oder auch andere Listen darstellen. Beachten Sie, daß Listenelemente nicht durch Kommas, sondern durch Leerzeichen getrennt werden.

Die Bearbeitung der Listen geschieht mit Funktionen, die ähnlich wie die BASIC-Funktion DEF FN Argumente verarbeiten und Ergebnisse liefern. Sie werden als

```
(func a b c d . . .)
```

definiert, wobei „func" den Namen der Funktion darstellt und a, b, c etc. die Argumente. Die Funktion bildet dabei das erste Element der Liste.

Eine Liste der ersten sechs Primzahlen wird folgendermaßen geschrieben:

```
(1 2 3 5 7 11)
```

Eine Addition (hier in BASIC):

```
1+2+3+5+7+11
```

arbeitet in LISP mit der PLUS-Funktion:

```
(PLUS 1 2 3 5 7 11)
```

und liefert das Ergebnis 29. Da die Funktion PLUS beliebig viele Argumente enthalten kann, ist auch folgendes möglich:

```
(PLUS 1 2 3 (PLUS 5 7 11))
```

Hier wird zuerst das PLUS mit der höchsten Schachtelungstiefe bewertet. Das Ergebnis 23 wird dann von dem äußeren PLUS als viertes Argument eingesetzt – als Ergebnis erscheint 29. Wichtig ist die Leichtigkeit, mit der sich Funktionen verschachteln lassen. Mit der Funktion SETQ ordnen wir das Ergebnis der Variablen A zu:

```
(SETQ A (PLUS 1 2 3 5 7 11))
```

SETQ hat dabei zwei Argumente – die Variable A und die Funktion, die die Ganzzahl 29 ergibt. SETQ ist natürlich selbst eine Funktion und liefert wiederum ein Ergebnis (hier den Wert 29). Die Zuordnung:

```
LET B = 1+2+3+5+7+11
```

Trotz des wachsenden Interesses an LISP gibt es nur wenige preiswerte Sprachversionen für Microcomputer. Das Acornsoft-LISP läuft auf dem Acorn B und dem Acorn Electron.

Für das Betriebssystem CP/M sind mehrere LISP-Versionen erhältlich, darunter Toolworks LISP/80, iLISP und muLISP-83 – die CP/M-Version von Microsoft.

Die mit CP/M arbeitenden Interpreter sollten auf allen CP/M-Micros laufen. Toolworks bringt 3600 Listenzellen und 11000 Namenszeichen für Atome in 48 KByte unter. Alle Programme sind über Gray Matter Ltd., 4 Prigg Meadow, Ashburton, Devon TQ13 7DF, Großbritannien, zu erhalten.

ist den meisten BASIC-Versionen nicht möglich, läßt sich in LISP aber als:

```
(SETQ B (SETQ A (PLUS 1 2 3 5 7 11)))
```

schreiben. Durch das Einsetzen der Funktion TIMES kann der doppelte Wert von A der Variablen B zugeordnet werden:

```
(SETQ B (TIMES 2 (SETQ A (PLUS 1 2 3 5
7 11))))
```

Dieser Ausdruck bewertet zuerst das PLUS und errechnet den Wert 29. Dieser wird von der inneren SETQ-Funktion der Variablen A zugeordnet, die das Ergebnis (29) als zweites Argument an die TIMES-Funktion weiterreicht. Das neue Ergebnis (58) wird nun an die äußere SETQ-Funktion übergeben, die diesen Wert in die Variable B setzt. Das Ergebnis des Ausdrucks ist also der Wert 58, der in weiteren Funktionen verwendet werden kann.

Selbständige Klammerauflösung

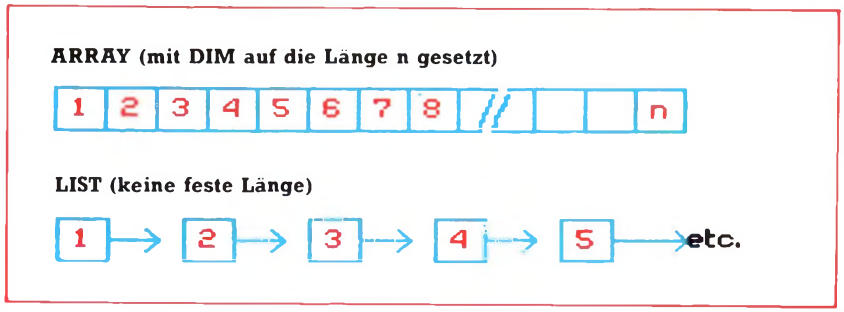
Aufgrund der vielen Klammern wird der Ausdruck sehr unübersichtlich. In einem sorgfältig aufgebauten Programm erledigt sich die Auflösung der Klammern jedoch fast von selbst. Einige LISP-Systeme geben hier Hilfestellung, indem sie über die Anzahl der offenen Klammern informieren. Das Acornsoft-LISP zeigt beispielsweise am Zeilenanfang durch Pfeile an, wie viele Klammern noch geschlossen werden müssen, bevor der Ausdruck vollständig ist.

In unserem letzten Beispiel hatte TIMES zwei Argumente: die Ganzzahl 2 und einen Listenausdruck, der zuvor auf das Ergebnis 29 gesetzt wurde. TIMES kann aber wie PLUS beliebig viele Argumente besitzen. Jedes der folgenden Beispiele ist legal und ergibt 1024:

```
(TIMES 1 2 4 8 16)
(TIMES 1 2 4 8
(TIMES 4 4))
(TIMES 1 2 (TIMES 2 2) (TIMES 2 4)
(PLUS 8 8))
```

In der Praxis begrenzen die meisten LISP-Systeme jedoch die Anzahl der Funktions-Argumente. Das Acornsoft-LISP unterstützt beispielsweise nur 28 Argumente, während andere Versionen die Anzahl noch mehr einschränken.

Die Beispiele zeigen deutlich, daß LISP-Befehle nur aus Argumentenlisten bestehen, in denen das erste Element die auszuführende Funktion angibt, während die darauf folgenden Elemente Argumente dieser Funktion sind. Diese Argumente können wiederum Listen sein, deren erste Elemente



als Funktionen Ergebnisse liefern. Was geschieht jedoch, wenn wir keine Funktion benötigen, sondern nur eine Liste mit Datenelementen, die wir als Titel einsetzen wollen? Wir können

```
(COMPUTER KURS LISP SERIE)
```

schlecht als Liste mit vier Datenelementen anlegen, da LISP versuchen würde, COMPUTER als Funktionsname zu bewerten und die Argumente (KURS, LISP und SERIE) als drei Variablennamen ansieht. Wir können LISP jedoch durch ein voranstehendes (') mitteilen, daß der Ausdruck nicht bewertet werden soll:

```
'(COMPUTER KURS LISP SERIE)
```

läßt sich dann wiederum als Liste mit vier nicht-numerischen Elementen der Variablen MAG zuordnen:

```
(SETQ MAG '(COMPUTER KURS LISP
SERIE))
```

Beachten Sie, daß es in LISP keine unterschiedlichen Variablentypen gibt. Die folgenden Ausdrücke sind in LISP daher alle völlig legal:

```
(SETQ A 3)              A = die Ganzzahl 3
(SETQ A 'COM-           A = der String 'COM-
  PUTER)                PUTER'
(SETQ A                 A = das Ergebnis von
  (PLUS 2 4 8))         2+4+8
(SETQ A '(1 2 4 8))     A = die Liste (1 2 4 8)
(SETQ A '(COM-          A = die LISTE (COM-
  PUTER KURS))          PUTER KURS)
(SETQ A B)              A = der Wert der
                        Variablen B
```

Es gibt nur wenige LISP-Versionen, die mit Fließkommaarithmetik arbeiten. In den meisten Fällen reichen die Ganzzahlen aus, wobei sich Fließkommaberechnungen ohne größere Schwierigkeiten durch normale Ganzzahlen simulieren lassen.

In der nächsten Folge werden wir sehen, wie LISP-Funktionen die Datenargumente bearbeiten und wie flexibel die Sprache bei Recursionen ist.

Listen haben Arrays gegenüber zwei große Vorteile. Zunächst muß vor ihrem Einsatz kein Speicherplatz für sie reserviert werden. Da sie dynamische Strukturen sind (das heißt keine feste Länge haben), können sie sich während der Programmausführung an die Menge der Daten anpassen. Listen lassen sich außerdem bei der Programmierung Künstlicher Intelligenz für recursive Prozeduren einsetzen.

Funktionelles

Wir zeigen, wie sich innerhalb von Listen andere Listen anlegen lassen, und definieren eigene Funktionen.

In der ersten Folge hatten wir untersucht, wie LISP veranlaßt wird, eine Liste als Datenliste und nicht als Funktion anzusehen. So ordnet

```
(SETQ X '(A B C D E F))
```

der Variablen X die Liste (A B C D E F) zu, wobei jedes Listenelement ein String mit einem einzigen Zeichen ist. Was aber passiert, wenn D, E und F Strings mit einzelnen Zeichen wären, A, B und C aber Variablen mit den Werten 2, 4 und 8? Für diesen Fall müssen wir X die Liste (2 4 8 D E F) zuordnen. Dies geschieht mit der neuen Funktion LIST, die eine Liste ihrer Argumente aufbaut.

```
(SETQ X '(LIST 'A 'B 'C 'D 'E 'F))
```

bringt das gleiche Ergebnis wie das vorige Beispiel, während:

```
(SETQ X '(LIST A B C 'D 'E 'F))
```

die Liste (2 4 8 D E F) zuordnet. Durch das Fehlen der Anführungsstriche werden A, B und C bewertet. Ebenso weist

```
(SETQ X (LIST 1 2 4 '(PLUS 4 4))
```

der Variablen X die Liste (1 2 4 (PLUS 4 4) zu, wobei das vierte Listenelement selbst wiederum eine Liste mit den drei Elementen PLUS, 4 und 4 ist. PLUS wurde nicht bewertet, da ein Anführungszeichen vor dem Ausdruck steht.

```
(SETQ X (LIST 1 2 4 (PLUS 4 4))
```

ordnet X jedoch die Liste (1 2 4 8) zu, da PLUS bewertet wurde. Ein einziges Anführungszeichen erzeugt diesen Unterschied. Mit dem Konzept läßt sich auch eine Liste von Listen aufbauen:

```
(SETQ PERSON '((KLAUS BLOCK)  (17 1 1960)))
```

ordnet der Variablen PERSON eine Datenliste zu, deren zwei Elemente wiederum Listen sind. Die erste Liste enthält zwei Elemente, die aus Zeichen bestehen, und die zweite Liste drei numerische Elemente. Innerhalb der Liste lassen sich weitere Listen anlegen.

Bisher haben wir uns die Funktionen SETQ, LIST, PLUS und TIMES angesehen. LISP verfügt jedoch noch über viele weitere Funktionen, deren Einsatz von der Anwendung abhängt.

Bevor wir uns ansehen, wie Funktionen angelegt werden, untersuchen wir zunächst drei Grundfunktionen, die in allen LISP-Versionen zu finden sind: CAR, CDR und CONS. Der Name CONS steht für das Wort CONStruct, während CAR und CDR aus einer der ersten Anwendungen stammen und Abkürzungen für „Contents of Address Register" und „Contents of Decrement Register" sind.

CAR und CDR

CAR und CDR müssen in jedem Fall ein einziges Argument haben, das eine Liste mit mindestens einem Element sein muß. Ausgeschlossen ist die „leere Liste" oder NIL, die

```
()
```

geschrieben wird. Die Funktion CAR liefert als Ergebnis das erste Element des Listenarguments.

```
(CAR '(1 2 3 4 5))
```

hat daher als Ergebnis die ganze Zahl 1. Das Ergebnis kann aber auch eine Liste sein. In dem Ausdruck

```
(CAR '((1 2) (3 4) 5))
```

ist das erste Element die Liste (1 2).

Die Funktion CDR liefert alle Elemente einer Liste außer dem ersten. So ergibt

```
(CDR '(1 2 3 4 5))
```

die Liste (2 3 4 5) und

```
(CDR '((1 2) (3 4) 5))
```

die Liste ((3 4) 5).

Die Funktion CONS fügt das erste Argument an das zweite an.

```
(CONS '(1 2) '((3 4) 5))
```

ergibt daher die Liste ((1 2) (3 4) 5).

Die Funktionen CAR und CDR werden oft ineinander verschachtelt. So ergibt

```
(CDR '((1 2) (3 4) 5))
```

die Liste ((3 4) 5) und

```
(CAR (CDR '((1 2) (3 4) 5)))
```

(3 4) – eigentlich (CAR '((3 4) 5)) –, während

```
(CAR (CAR (CDR '((1 2) (3 4) 5))))
```

den Wert 3 liefert – (CAR '(3 4)).

Da diese Verschachtelungen recht lästig werden können, erlaubt LISP Abkürzungen. Die Funktionsnamen fangen dabei zwar immer noch mit C an und enden mit R, können dazwischen aber jede beliebige Anzahl As (für CAR) oder Ds (für CDR) enthalten. Das Acornsoft-LISP läßt jedoch nur bis zu drei Einfügungen zu. Der letzte Ausdruck ließe sich nun auch

```
(CAAR (CDR '((1 2) (3 4) 5)))
```

schreiben oder

```
(CAR (CADR '((1 2) (3 4) 5)))
```

oder auch

```
(CAADR '((1 2) (3 4) 5))
```

Alle Versionen liefern das Ergebnis 3.

Funktionen werden mit der Funktion DEFUN definiert, die drei Argumente braucht und folgendes Format hat:

```
(DEFUN a (b) (c))
```

Dabei ist a der Name der Funktion, b die Parameterliste (ähnlich wie in PASCAL oder Acorn-B-BASIC) und c eine Listenstruktur mit dem Hauptteil der Funktion. Wenn wir beispielsweise oft mit 8 multiplizieren müssen, könnten wir natürlich jedesmal schreiben:

```
(TIMES 8 N)
```

wobei N die zu multiplizierende Zahl darstellt. Eine selbstdefinierte Funktion könnte jedoch die gleiche Aufgabe ausführen:

```
(DEFUN TIME8 (N) (TIMES 8 N))
```

Die Funktion TIME8 nimmt N als Argument, multipliziert diese Zahl über die Standardfunktion mit 8 und liefert das Ergebnis.

```
(TIME8 11)
```

ergibt 88.

Bevor wir weitere Funktionen definieren, wollen wir jedoch erst einmal ein wichtiges Programmierkonzept untersuchen: die Bedingung (COND). Sie ist etwa so wichtig wie der IF...THEN-Befehl in BASIC.

Auch Bedingungen haben in LISP die Form von Funktionen:

```
(COND (Bedingung1 Ausdruck1)
      (Bedingung2 Ausdruck2)
            "     "
      (BedingungX AusdruckX))
```

Beachten Sie, daß diese Funktion sich über mehrere Zeilen erstrecken muß, um überhaupt ausreichend Platz zu haben. Diese Anordnung wird in LISP oft eingesetzt, da die Sprache an der letzten Klammer erkennt, wo das Ende des Ausdrucks liegt.

```
(DEFUN EQUAL (A B) (COND
   ((EQ A B) T)
   ((OR (ATOM A) (ATOM B)) NIL)
   ((EQUAL (CAR A) (CAR B)) (EQUAL (CDR A) (CDR B))) (T NIL)))
(DEFUN FIND_AUTO (X L) (COND
   ((NULL L) '(KEINE AUTONUMMER))
   (EQUAL X (CDAR L)) (CAR L))
   (T (FIND_AUTO X (CDR L)))))

(SETQ AUTOS '((SCHMITT HHCS1254) (MEIER PIOS2354)
              (SCHANZE BIDD1984) (STEIN BTT336))
(PRINTC (FIND_AUTO '(BIDD1984) AUTOS))
```

Dieses einfache Programm zum Auffinden von Daten zeigt den Einsatz der Funktionen CDR und EQ. Dabei liefert eine vom Anwender eingegebene Autonummer – falls in der Datenbank des Programms vorhanden – diese Nummer und den Nachnamen des Besitzers.

Die COND-Funktion ähnelt dem CASE-Befehl von PASCAL. Dabei wird Bedingung1 bewertet. Ist sie wahr, dann liefert Ausdruck1 das Ergebnis. Ist sie falsch, wird Bedingung2 getestet etc. Einige LISP-Versionen (auch Acornsoft-LISP) verlangen, daß zumindest eine der Bedingungen („Aussagen" genannt) wahr ist. Dies läßt sich jedoch leicht durch das Hinzufügen einer Endbedingung erreichen:

```
(COND (Bedingung1 Ausdruck1)
      (Bedingung2 Ausdruck2)
            "     "

(COND (T    Letzter Ausdruck))
```

Das Zeichen T stellt hier „True" (wahr) dar. Der letzte Ausdruck (falls er erreicht wird) ist immer bewertbar. In LISP ist:

```
T = True = Nicht Null
```

und

```
F = False = Null = ()
```

Die Klammern stellen die leere Liste dar. Wir können nun schon kompliziertere Funktionen definieren.

Fast alle BASIC-Dialekte kennen die Funktion ABS, die als Ergebnis den positiven Wert ihres Arguments liefert. Das heißt: ein positives Argument bleibt unverändert, ein negatives wird umgekehrt. In LISP hat diese Funktion folgendes Format:

```
(DEFUN ABS (X)
   (Funktionsinhalt))
```

wobei X ein ganzzahliges Argument ist. Mit der Bedingungsfunktion COND sieht die vollständige ABS-Funktion folgendermaßen aus:

```
(DEFUN ABS (X)
   (COND ( (MINUSP X) (MINUS X))
         ( T                X )))
```

Die Definition enthält zwei neue Funktionen. Die Testfunktion MINUSP liefert einen Wert, wenn das Argument eine negative Zahl ist. Bei einer positiven Zahl ist das Ergebnis „False". Ist die Bedingung wahr, negiert die Funktion MINUS ihr Argument, wenn nicht, liefert T (die immer wahr ist) den ursprünglichen Wert.

Testfunktionen
Die folgende Liste zeigt Testfunktionen, die in LISP oft eingesetzt werden:

(OR Arg1 Arg2 ...)	Liefert „True", wenn mindestens eins der Argumente wahr ist
(ONEP Arg)	Liefert „True", falls Arg = 1
(ZEROP Arg)	Liefert „True", falls Arg = 0
(LISTP Arg)	Liefert „True", falls Arg eine Liste ist, und „false", wenn es ein Atom ist
(ATOM Arg)	Liefert „True", falls Arg ein Atom ist, und „False", wenn es eine Liste ist
(LESSP Arg1 Arg2)	Liefert „True", wenn Arg1 < Arg2
(GREATERP Arg1 Arg2)	Liefert „True", falls Arg1 > Arg2
(EQ Arg1 Arg2)	Liefert „True", falls Arg1 = Arg2

Königliches Spiel

In der letzten Folge unserer LISP-Serie beschäftigen wir uns mit weiteren Standardfunktionen der Sprache und führen sie am Beispiel von zwei komplexen Problemen vor. Zunächst untersuchen wir jedoch, wie LISP Datenstrukturen darstellt.

LISP verwendet „Pointer" (Zeiger) und bezieht sich so mit einer einheitlichen Schreibweise auf Namen, Zeichen, Zahlen und Listen. Pointer lassen sich in jeder Sprache – auch in BASIC – leicht simulieren.

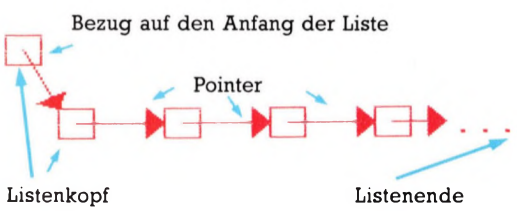
```
POKE 100,55 : POKE 99,100 (fast alle Micros)
```
oder
```
?100 = 55 : ?99 = 100 (Acorn B/Electron)
```

halten in den Speicherstellen 100 und 99 die angegebenen Werte. Der Wert in Adresse 99 läßt sich als Pointer auf Speicherstelle 100 verstehen. Bei dem Befehl

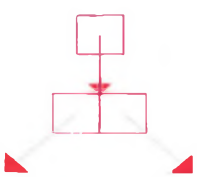
```
PRINT PEEK(PEEK(99))
```
oder
```
PRINT ?(?99)
```

erhalten wir die Antwort 55. Beachten Sie, daß sich die Speicherstellen 99 und 100 auf einigen Micros im ROM befinden und sich daher nicht ändern.

Natürlich können Pointer auf jeden Speicherbereich zeigen. Über 255 liegende Adressen brauchen jedoch zwei Bytes, um den Bereich zwischen 0 und 65535 umfassen zu können. Wenn wir den Wert 55 als Pointer auf eine weitere Speicherstelle ansehen und so weiter, erhalten wir folgende Listenstruktur:

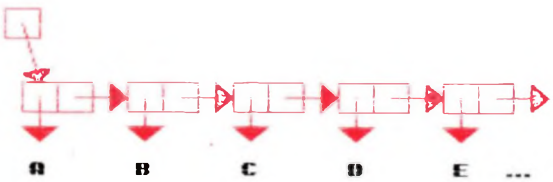


Diese Liste enthält noch keine Informationen. Wenn man jedoch zwei Adressen zusammenfügt, entstehen an jedem Knotenpunkt zwei Pointer:



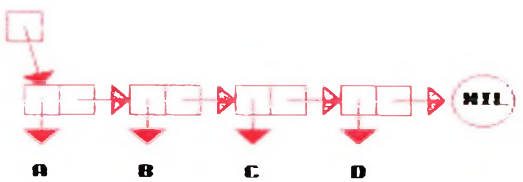
Dieses Verfahren läßt sich nun in ein Listenformat umsetzen:

```
(A B C D E . . .)
```



Nun fehlt nur noch ein Element, das das Listenende anzeigt. Dafür wird der letzte Pointer auf die leere Liste (oder NIL) gesetzt. Hier die Liste mit fester Länge:

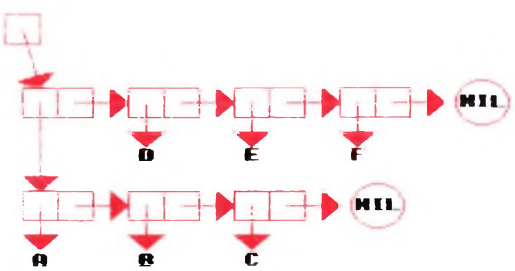
```
(A B C D)
```



Bei einer Liste im Innern einer anderen Liste zeigt nur einer der linken Pointer auf die zweite Liste statt auf ein Atom.

```
((A B C) D E F)
```

wird daher folgendermaßen dargestellt:



An dieser Stelle wird auch deutlich, wie die Funktionen CAR und CDR arbeiten. CAR (der Kopf des Argumentes) ist der linksstehende Pointer des ersten Argumentelements, während CDR (alle Elemente außer dem Kopf) der rechtsstehende Pointer des ersten Elements ist. Der Aufruf

```
CONS((CAR L) (CDR L))
```

liefert daher die ganze Liste.

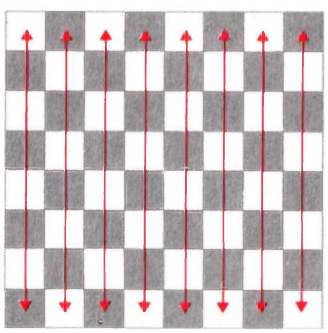
Mit CONS lassen sich Listen anlegen. Die Funktion baut aus seinen beiden Argumenten eine weitere Liste auf und liefert als Ergebnis einen Pointer auf das erste Element der neuen Liste. CONS kann jede Art von Liste aufbauen. Wenn die Zahl der CONS-Aufrufe zu groß wird, kann die Kurzfunktion LIST verwandt werden.

Neue Informationen sollten immer mit CONS an den Kopf einer Liste angefügt werden, da beim Anfügen an das Listenende zuerst die ursprüngliche Liste kopiert werden muß, damit der Endpointer auf das neue Element zeigt.

Einige LISP-Versionen führen diese Aufgabe mit APPEND automatisch aus. Diese Funktion ist jedoch nicht so schnell wie CONS. Aus Geschwindigkeitsgründen wird manchmal nur der Pointer der ursprünglichen Liste umgesetzt und keine Kopie angelegt.

Zum Abschluß dieser Serie stellen wir die Lösung für ein bekanntes Problem vor: Auf einem Schachbrett sollen acht Königinnen aufgestellt werden, ohne sich gegenseitig zu bedrohen. Offensichtlich muß in jeder Reihe und Spalte genau eine Königin stehen. Die Schwierigkeit liegt darin, die Reihen und Spalten so anzuordnen, daß sich die Reichweiten der Figuren in der Diagonale nicht überschneiden.

Jede Spalte des Schachbretts enthält eine Königin, die gemäß ihrem Platz von 1 bis 8 in einer Liste namens KOENIGINNEN gespeichert sind. Die Spalte jeder Königin (der Wert von KOENIGINNEN) ist eine Variable und wird von LISP berechnet.



KOENIGINNEN (K1 K2 K3 K4 K5 K6 K7 K8)

Beim Programmanfang ist die Liste KOENIGINNEN noch leer. Bei jedem Schritt wird eine neue Königin in Reihe 1 am Anfang der Liste eingesetzt. Die Position dieser Königin (und möglicherweise auch die der anderen) wird mit der Funktion AENDERN so lange verändert, bis alle Königinnen auf sicheren Positionen stehen. Die neue Liste wird dann wiederum zu KOENIGINNEN, an deren Anfang eine weitere Königin angefügt wird, und so weiter.

Dieser Vorgang setzt sich fort, bis die Liste acht Königinnen enthält. An diesem Punkt wird das Ergebnis mit PR_SCHACHBRETT gedruckt. Die Anfangsfunktion nennen wir LOESUNG. Sie sieht folgendermaßen aus:

```
(DEFUN LOESUNG ()
 (SETQ KOENIGINNEN ())
 (LOOP (UNTIL EQ (LAENGE (KOENI-
   GINNEN) 8)) (SETQ KOENIGINNEN (ALTER
   (CONS 1 KOENIGINNEN))))
 (PR_SCHACHBRETT ' (LT LS LL LK RK RL
   RS RT) KOENIGINNEN))
```

Hier ist leicht zu erkennen, wie LOESUNG anfangs die leere Liste KOENIGINNEN anlegt. Die Funktion LOOP setzt KOENIGINNEN auf den neuen Wert der mit AENDERN aktualisierten Liste. Dabei steht eine Königin in Reihe 1 gefolgt von der zuvor angelegten Liste. Die Schleife LOOP wird so lange ausgeführt, bis (UNTIL) die LAENGE von KOENIGINNEN dem Wert 8 gleicht (EQ). Danach wird die fertige Liste gedruckt.

Das Programm enthält drei neue Funktionen. LAENGE liefert die Anzahl der Elemente eines Arguments und ist folgendermaßen definiert:

```
(DEFUN LAENGE (L (N))
 (SETQ N 0)
 (LOOP (WHILE L N)
   (SETQ N (ADD1 N))
   (SETQ L (CDR L)) ))
```

Mit dem in Klammern gestellten Argument N stellt Acornsoft-LISP mögliche Argumente dar. In unserem Fall wird N nicht an die Funktion übergeben – es ist nur eine lokale Variable.

Die Funktion PR_SCHACHBRETT kombiniert beide Argumentenlisten zu einem Ausdruck:

```
(DEFUN PR_SCHACHBRETT (ZEILEN
 SPALTEN) (COND ((NULL ZEILEN) T)
   (T (PRINTC (CAR ZEILEN) (CAR SPALTEN))
    (PR_SCHACHBRETT (CDR ZEILEN) (CDR
      SPALTEN)) )))
```

Da beide Listen die gleiche Größe haben, wird die Funktion beendet, wenn die erste Bedingung eintritt (eine Liste ist leer). Falls nicht, wird das erste Element jeder Liste gedruckt (mit Return), während sich die Funktion für den Rest der Liste immer wieder selbst aufruft.

Die dritte Funktion AENDERN schließlich erledigt die Hauptarbeit:

```
(DEFUN AENDERN (KOENIGINNEN)
  (COND ((EQ (CAR KOENIGINNEN) 9)
    (AENDERN (CONS (ADD1
      (CADR KOENIGINNEN))
      (CDDR KOENIGINNEN) )))
    ((SICHER KOENIGINNEN) KOENI-
      GINNEN)
    (T (AENDERN (CONS (ADD1
      (CAR KOENIGINNEN))
      (CDR KOENIGINNEN) )))))
```

AENDERN überprüft, ob die zuletzt eingesetzte Königin vom Spielbrett „gefallen" ist (Zeile 9). Ist dies der Fall, müssen die Positionen der anderen Königinnen verändert werden.

Da die „gefallene" Königin alle Zeilen von 1 bis 8 überprüft hat (die dritte Bedingung von AENDERN), gibt es keine Möglichkeit, sie bei den augenblicklichen Positionen der anderen Königinnen auf dem Brett aufzustellen. AENDERN ruft sich daher für den Rest der Liste (alle Königinnen außer der ersten) selbst auf.

Dabei inkrementiert die Funktion die Zeile der letzten Königin. Dieser Vorgang wird „Backtracking" genannt. Er ist bei dieser Art von Computeralgorithmen weit verbreitet. Ist die zweite Bedingung TRUE, dann stehen alle Königinnen auf sicheren Positionen und die Liste kann zurückgegeben werden. Ist dies

nicht der Fall, wird die Position der zuletzt eingesetzten Königin inkrementiert und ein weiterer Versuch unternommen.

Nun fehlt nur noch die Funktion SICHER, die überprüft, ob alle Königinnen auf unangreifbaren Positionen stehen. Da in der Liste KOENIGINNEN die Spalten bereits festgelegt sind und zwei Königinnen nie die gleiche Position einnehmen können, brauchen nur die Zeilen und Diagonalen überprüft zu werden. Die Funktion KEINGARDEZ ergibt TRUE, wenn keine Königin eine andere bedroht. SICHER wird folgendermaßen definiert:

```
(DEFUN SICHER (KOENIGINNEN)
  (COND ((NULL KOENIGINNEN) T)
    (T (AND (KEINGARDEZ
       (CAR KOENIGINNEN)
       (CDR KOENIGINNEN) 1)
       (SICHER (CDR KOENIGINNEN)) ))))
```

Dieser Ablauf läßt sich in normalen Worten leichter verstehen:

„Die erste Königin der Liste KOENIGINNEN bedroht keine andere Königin der Liste UND alle anderen Königinnen der restlichen Liste sind sicher."

Nun muß nur noch die Funktion KEINGARDEZ definiert werden. Sie soll TRUE ergeben, wenn es in den Zeilen und Spalten keine Kreuzungspunkte gibt:

```
(DEFUN KEINGARDEZ (NEW REST SPALTE)
  (COND ((NULL REST) T)
    (T (AND (NOT (EQ NEW (CAR REST)))
      (NOT (EQ SPALTE (ABS (DIFFERENCE
        NEW (CAR REST)))))
      (KEINGARDEZ NEW (CDR REST)
        (ADD1 SPALTE)) ))))
```

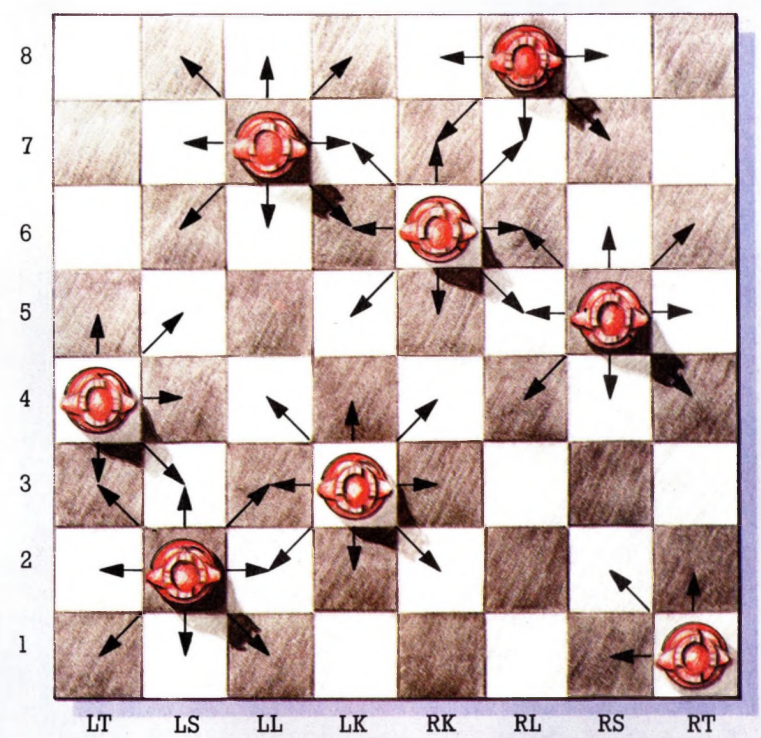
Hier ist NEW die zuletzt eingesetzte Königin und REST die Liste aller anderen Königinnen des Brettes. Angenommen, andere Königinnen sind auf dem Brett vorhanden (NULL REST ist FALSE), so ergibt die AND-Funktion TRUE, wenn:

1) die augenblickliche Zeile der Königin nicht die gleiche ist wie die Zeile der ersten Königin der restlichen Liste;

2) die Königinnen nicht auf der gleichen Diagonalen liegen. Bei der Überprüfung wird sichergestellt, daß die absolute Zeilendifferenz nicht die gleiche ist wie die Spaltendifferenz. SPALTE enthält dabei die Zahl der Spalten, die zwischen der augenblicklichen Königin und der nächsten Königin der Liste liegen (anfangs 1). ABS wurde schon in einer früheren Folge definiert:

```
(DEFUN ABS (ZAHL)
  (COND ((MINUSP ZAHL) (MINUS ZAHL))
    (T ZAHL) ))
```

3) Punkt 1 und 2 für die augenblickliche Königin und auch die nächste Königin der Liste zutreffen. Dies wird überprüft, indem die Funktion sich selbst aufruft, dabei aber nur die restlichen Königinnen mit Ausnahme des ersten Elements anspricht und die Spaltendifferenz inkrementiert.

Jetzt sind alle Definitionen vorhanden. Das Bild zeigt die Losung.

Wahrhaft königlich
Das „Problem der acht Königinnen" liefert ein perfektes Beispiel für Rätsel, die sich mit den Listen und recursiven Funktionen von LISP schnell lösen lassen. Probieren Sie es selbst einmal, bevor Sie sich die nebenstehende Lösung ansehen. Setzen Sie acht Königinnen derart auf ein Schachbrett, daß sie sich nicht gegenseitig bedrohen. Sie brauchen Schach nicht zu kennen, sondern müssen nur wissen, daß Königinnen sich in gerader Linie beliebig weit diagonal, vertikal und horizontal bewegen können und dabei jede im Weg liegenden Figur schlagen.